UNIVERSIDADE DA BEIRA INTERIOR

Algoritmos e Estruturas de Dados **2º Semestre**

Exame – Época Recurso (16 val) **2h + 15min** **05/07/2024**

A. [1.50 val] Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

```
for (k = 1; k < n; k = k + 1) {
   for (j = n; j > k; j = j - 1)
      T = k + j;
}
```

1. Simule o bloco de código dado, considerando que **n** é um número inteiro **muito grande**. Apresente os dados da simulação numa tabela cujo cabeçalho é o seguinte:

k	k < n (V/F)	j	j > k (V/F)	T = k + j
…	…	…	…	…

2. A partir dos resultados da simulação obtidos em 1, determine a **ordem de complexidade** do algoritmo associado ao bloco de código dado em cima. Incidir os cálculos sobre a operação que mais vezes é executada ("**j > k**"). Justifique, apresentando os cálculos efetuados.

B. [2.00 val] Listas ligadas

Considere as seguintes declarações:

```
struct Nodo {
   int   Elemento;
   struct  Nodo  *Prox;
};
typedef  struct  Nodo *PNodo;
```

Considere também os seguintes protótipos de funções já implementadas:

```
int listaVazia (PNodo L);   // devolve 1 se a lista L é vazia e 0 caso contrário
PNodo libertarNodo (PNodo P);   // liberta a zona de memória apontada por P e devolve NULL
```

Usando ou **não** as funções fornecidas em cima, mas **não podendo** usar outras funções, **implemente uma função iterativa (não recursiva) em C** que

- **receba** uma lista **L** de elementos do tipo **inteiro** e um inteiro **X**
- **devolva** a lista **L** após a **remoção** do nodo de L com valor no campo **Elemento** maior do que X que se encontra mais próximo do início de L (ou seja, primeiro nodo com valor no campo Elemento maior do que X); ter em conta que a lista L pode estar vazia ou pode não ter elementos maiores do que X.

Nota: Otimizar o algoritmo, de forma a que a lista L seja percorrida apenas uma vez.

C. [2.00 val] EAD Pilha

Considere as seguintes declarações de EAD Pilha de elementos do tipo inteiro:

```
struct NodoPilha { … };
typedef  struct  NodoPilha *PNodoPilha;
```

e os seguintes protótipos de funções (operações básicas):

```
PNodoPilha criarPilha ();                  PNodoPilha push (int X, PNodoPilha S);
PNodoPilha pop (PNodoPilha S);             int topo (PNodoPilha S);
int pilhaVazia (PNodoPilha S);    // 1 (se vazia) ou 0 (se não vazia)
```

Usando as operações básicas sobre uma EAD Pilha, implemente uma **função em C** que

- **receba** uma **pilha S** já preenchida com valores inteiros positivos e um inteiro positivo **k** (k > 0),

- **devolva** dois resultados:
  - o elemento do **fundo** da pilha **S**, caso exista (se **S** está **vazia**, então devolver **-1**), e
  - a pilha **S** apenas com **k** elementos **mais próximos do fundo** de S, mas **mantendo a ordem inicial** dos elementos em S (se **k ≥ tamanho(S)**, então devolver **S**)

D. [1.50 val + 2.00 val] EAD Árvore Binária de Pesquisa (ABP)

Considere a seguinte declaração de EAD Árvore Binária de Pesquisa:

```
struct NodoABP {
   int    Elemento;
   struct  NodoABP  *Esquerda;
   struct  NodoABP  *Direita;
};
typedef  struct  NodoABP *PNodoABP;
```

1. Implemente (pode usar a função alturaABP) **uma função recursiva em C** que
   - **receba** uma ABP **T**,
   - **devolva** a quantidade de elementos (nodos) de **T** cujo valor no campo **Elemento** é **positivo não nulo e par** (> 0 e par).
2. Implemente **uma função iterativa (não recursiva) em C** que
   - **receba** uma ABP **T**
   - **devolva** o elemento (valor inteiro) com o maior valor negativo entre todos os elementos de T (se T está vazia ou se todos os elementos de T são não negativos (≥ 0), então deve devolver **1**).

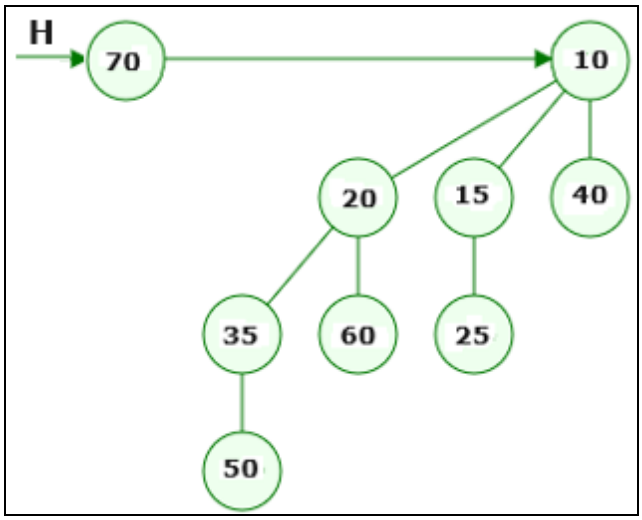
Nota: otimize os algoritmos criados (código escrito) tendo em conta a **definição de ABP**.

E. [2.50 val] ABP Balanceadas/Equilibradas (AVL)

1. **Construa** uma **ABP do tipo AVL inserindo** os nodos com os seguintes valores (um a um e pela ordem apresentada): **60**, **30**, **10**, **45**, **50**, **70** e **55**. (**NOTA**: não é necessário colocar as alturas). Sempre que **inserir um nodo**, deve **redesenhar** a **ABP** obtida (acrescentar o nodo à árvore) e **verificar se continua balanceada/equilibrada**. Caso **não fique equilibrada**, deve
   1º) indicar o nodo onde se deteta o desequilíbrio e o tipo de rotação a aplicar para reequilibrá-la,
   2º) reequilibrar a árvore e redesenhá-la (apresentando todos as árvores intermédias obtidas)
2. **Remova** o nodo da **raiz** da ABP **obtida em 1.**, **redesenhe** a árvore obtida, **verifique** se continua equilibrada e **reequilibre-a**, caso seja necessário. Apresente a árvore final.

F. [1.50 val] Heaps (filas de prioridade)

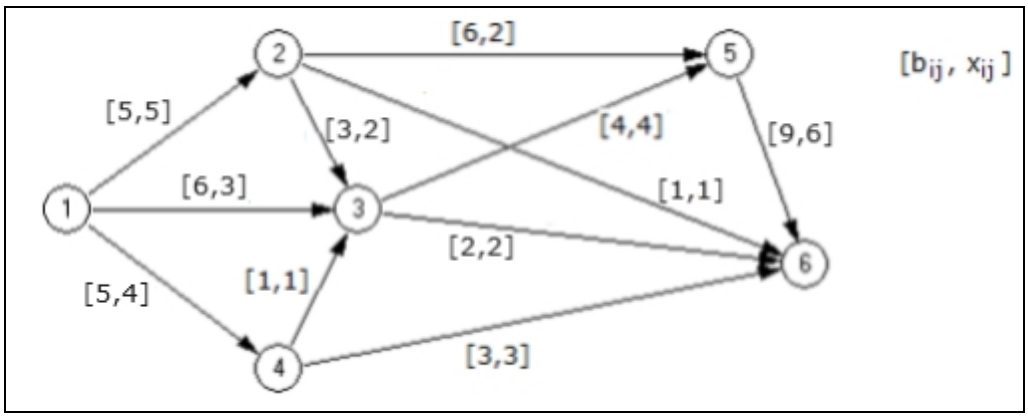
Considere o seguinte **heap Binomial H** com estrutura de minHeap:



1. Determine o resultado da operação **inserir** um elemento com o valor **55** no heap **H**. Justifique cada passo e apresente a evolução do heap desde a estrutura inicial (figura ao lado) até à final. Descreva o processo.
2. Determine o resultado da operação **"remover"** aplicado ao heap **H** (figura ao lado). Justifique cada passo e apresente a evolução do heap H desde a estrutura inicial (figura ao lado) até à final. Descreva o processo.

G. [3,00 val] Grafos

Considere a seguinte rede G = (A, N, C), onde $\mathbf{b_{ij}}$ é a capacidade e $\mathbf{x_{ij}}$ o fluxo do arco (i, j):



1. Determine o fluxo atual da rede entre os nós 1 e 6. Justifique.

2. Determine o fluxo que atualmente sai do nó **4**. Determine o fluxo que chega ao nó **5**. Justifique.

3. Determine o **fluxo máximo** que pode ser enviado do nó **1** para o nó **6**, aplicando o algoritmo de **Ford-Fulkerson**. Simule este algoritmo e apresente os dados da simulação.

Apresente os dados da simulação, em especial: indicação dos **nós rotulados** e cálculo dos respetivos **rótulos**, indicação dos **nós rotulados e varridos**, evolução dos valores de **j** e de **k**.

Esquema proposto para apresentar os dados da simulação:

	1	2	3	4	5	6
R(k)						

Nós **rotulados** ←

Nós **rotulados e varridos** ←

j ← ?

k ← …

k ← …

Algoritmo de Ford-Fulkerson:

Passo 1.

R(S) ← [S^+, ∞] (S é rotulado com S^+ e ∞)

S fica rotulado e não varrido

Passo 2. (processo de rotulação)

j (rotulado e não varrido) ← [i^+, Q(j)] ou [i^-, Q(j)]

para (todo **k** ∈ **N** não rotulado, tal que (j, k) ∈ A e $x_{jk} < b_{jk}$) **fazer**

R(k) ← [j^+, Q(k)], com Q(k) = min{ Q(j), $b_{jk} - x_{jk}$ }

fim_para

para (todo **k** ∈ **N** não rotulado, tal que (k, j) ∈ A e $x_{kj} > 0$) **fazer**

R(k) ← [j^-, Q(k)], com Q(k) = min{ Q(j), x_{kj} }

fim_para

j fica rotulado e **varrido**

todos os nós **k** ficam rotulados e não varridos

se (T está rotulado) **ou** (não é possível rotular T) **então**

(foi determinado um c.a.f. ⇒ **passar** ao **Passo 3**) **ou**

(não existe c.a.f. — o fluxo atual é máximo ⇒ **PARAR**)

senão

regressar ao **Passo 2** (início)

fim_se

Passo 3. (mudança de fluxo)

como (R(T) = [k^+, Q(T)]) **então**

x_{kT} ← x_{kT} + Q(T)

enquanto (k ≠ S) **fazer**

se (R(k) = [j^+, Q(k)]) **então**

x_{jk} ← x_{jk} + Q(T)

fim_se

se (R(k) = [j^-, Q(k)]) **então**

x_{kj} ← x_{kj} – Q(T)

fim_se

k ← j

fim_enquanto

apagar os rótulos

regressar ao **Passo 1**